ECS 271 - Machine Learning
Homework 2- Write-up
Playing Pong with Deep Q-Network

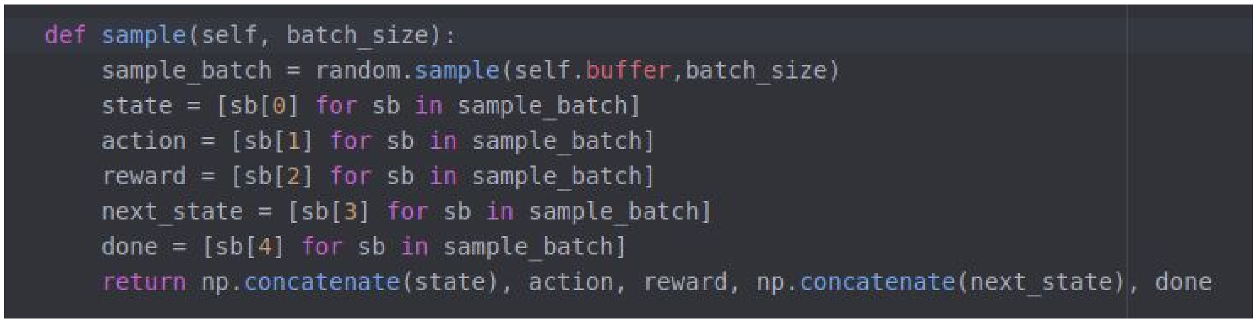
Name: **Bhargav Sundararajan**
Student ID: **916249586**

PART 1:

Q1: Explain what is a replay memory/buffer in DQN. Why it is necessary?
A: A replay memory is used to store the **model's experiences** at each time step e_t. The State (s_t), action performed (a_t), reward (r_t), and the next state (s_{t+1}) are stored respectively for each time step. At each iteration (after an initial number of skips), a batch of experiences is samples randomly from the replay memory and fed to the model. This step helps us calculate the loss at each epoch and help alter the weights and biases.
Importance of replay memory:

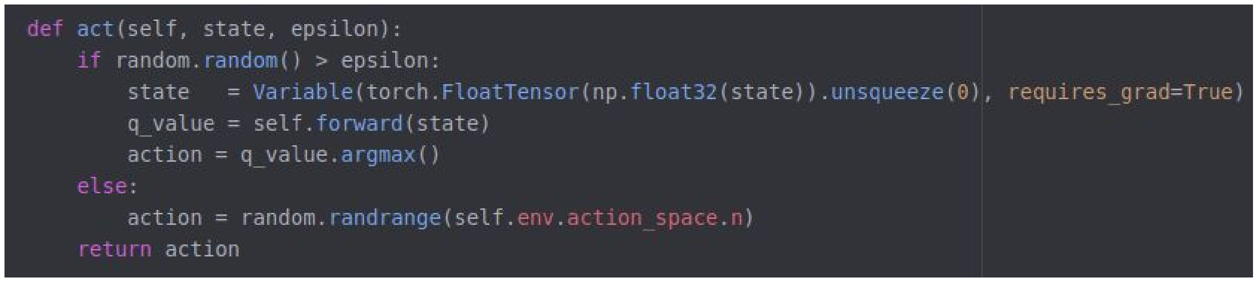
- Plays a crucial role in weight updates based on the loss calculated using the experiences of the replay memory.
- The replay memory helps in breaking correlations between consecutive samples and instead randomizes the input to the model. This helps the model as it reduces the variance of weight updates and also the model learns a more general policy.
- It also avoid oscillations by averaging out the behavior distribution and in turn smoothening the learning of the model.

Q2: Implement the "randomly sampling" function of replay memory/buffer:



```
def sample(self, batch_size):
    sample_batch = random.sample(self.buffer,batch_size)
    state = [sb[0] for sb in sample_batch]
    action = [sb[1] for sb in sample_batch]
    reward = [sb[2] for sb in sample_batch]
    next_state = [sb[3] for sb in sample_batch]
    done = [sb[4] for sb in sample_batch]
    return np.concatenate(state), action, reward, np.concatenate(next_state), done
```

Q3: Given a state, write code to calculate the Q value and the corresponding chosen
action based on neural network:

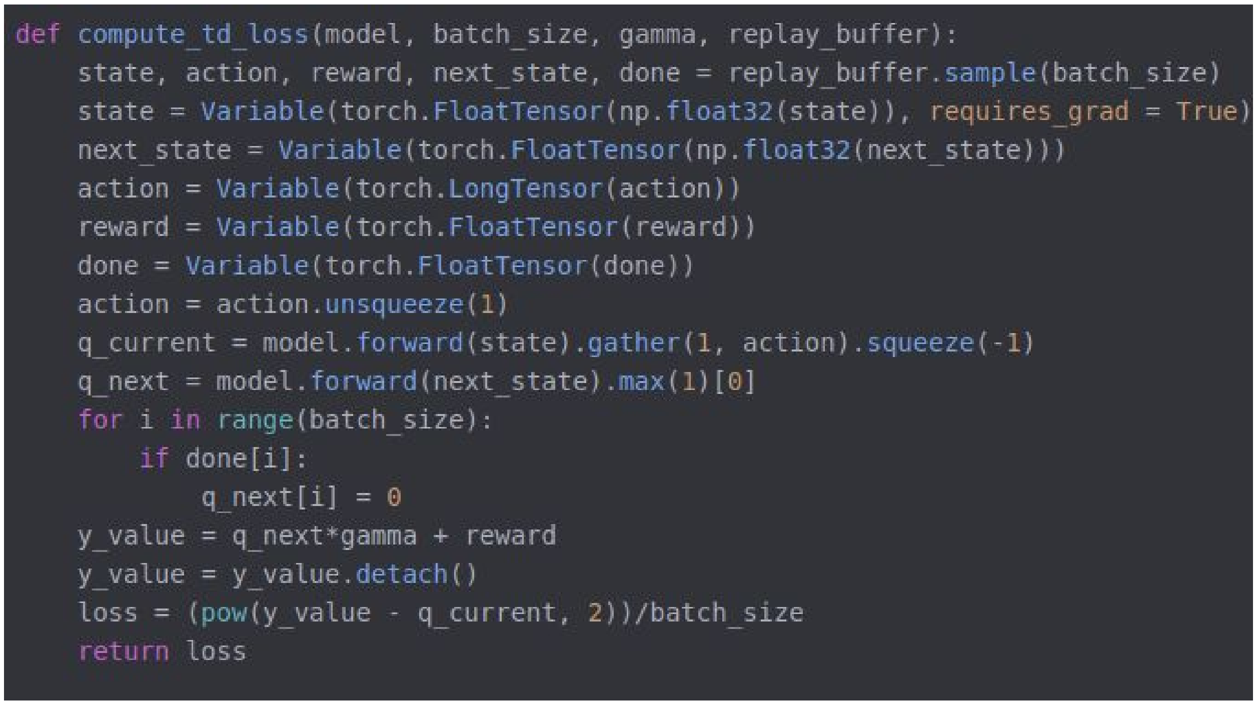


```
def act(self, state, epsilon):
    if random.random() > epsilon:
        state   = Variable(torch.FloatTensor(np.float32(state)).unsqueeze(0), requires_grad=True)
        q_value = self.forward(state)
        action = q_value.argmax()
    else:
        action = random.randrange(self.env.action_space.n)
    return action
```

Q4: Given a state, what is the goal of line 48 and line 57 of dqn.py ? Aren't we calculating the Q value and the corresponding chosen state from the neural network?

A: In the initial stages of training, the model would not have learned a good policy. If the actions are chosen based on Q-values alone in the initial stages, then the model will not explore much to learn a better policy. This can lead to underfitting and the model can get stuck at a local minimum. The epsilon greeedy approach will ensure that this does not happen by assigning random actions to a high number of initial states. As the model starts learning a better policy, the value of epsilon is decayed gradually to favor the Q-values to take the decision.

Q5: Implement the "Temporal Difference Loss": the objective function of DQN:



```
def compute_td_loss(model, batch_size, gamma, replay_buffer):
    state, action, reward, next_state, done = replay_buffer.sample(batch_size)
    state = Variable(torch.FloatTensor(np.float32(state)), requires_grad = True)
    next_state = Variable(torch.FloatTensor(np.float32(next_state)))
    action = Variable(torch.LongTensor(action))
    reward = Variable(torch.FloatTensor(reward))
    done = Variable(torch.FloatTensor(done))
    action = action.unsqueeze(1)
    q_current = model.forward(state).gather(1, action).squeeze(-1)
    q_next = model.forward(next_state).max(1)[0]
    for i in range(batch_size):
        if done[i]:
            q_next[i] = 0
    y_value = q_next*gamma + reward
    y_value = y_value.detach()
    loss = (pow(y_value - q_current, 2))/batch_size
    return loss
```

Q6: Training the Model:

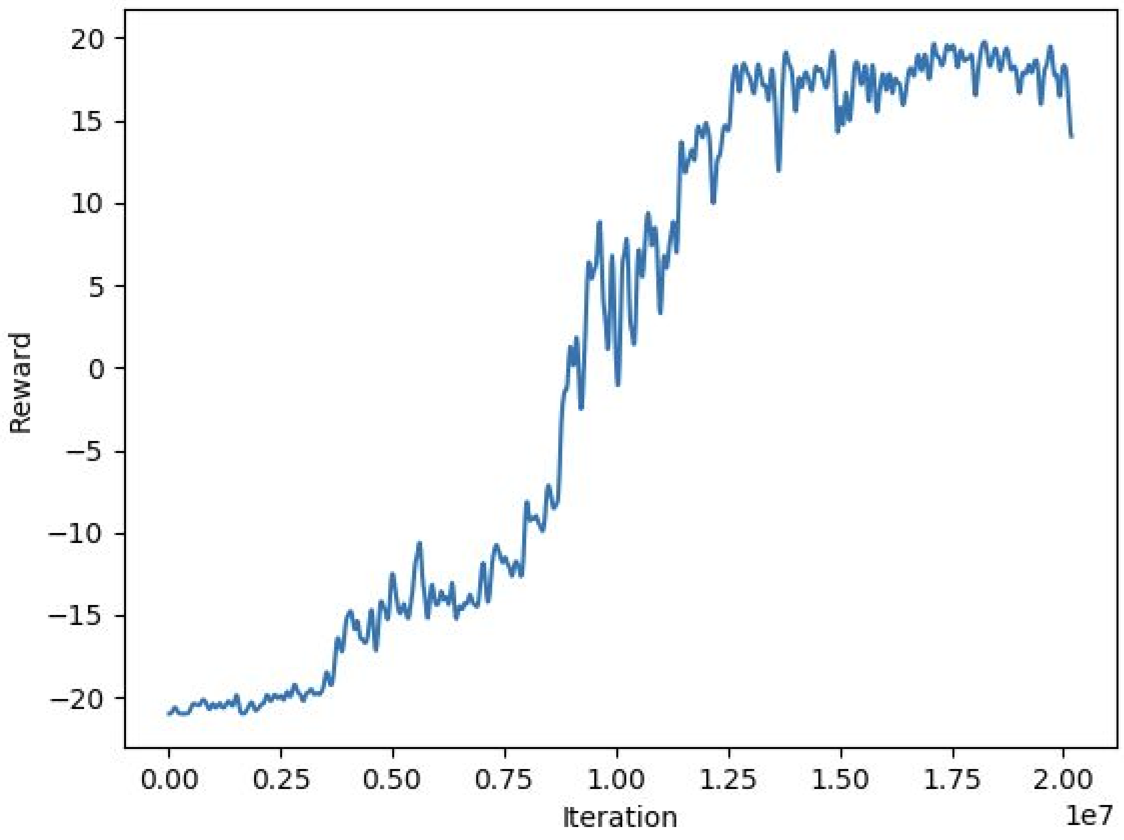
A: I had trained 3 models with slightly altered values of gamma (γ), Total number of frames and Length of replay memory.

MODEL 1:

For the first model, I doubled the Number of frames to train and the replay memory from the default setting. I chose to keep Gamma the same as my investigations suggested that it was the best value.

Gamma : 0.99
Number of frames: 2 Million
Length of Replay Memory: 200,000

The trend of the Rewards while training is as follows:



Maximum reward: 21
Average last reward: 20

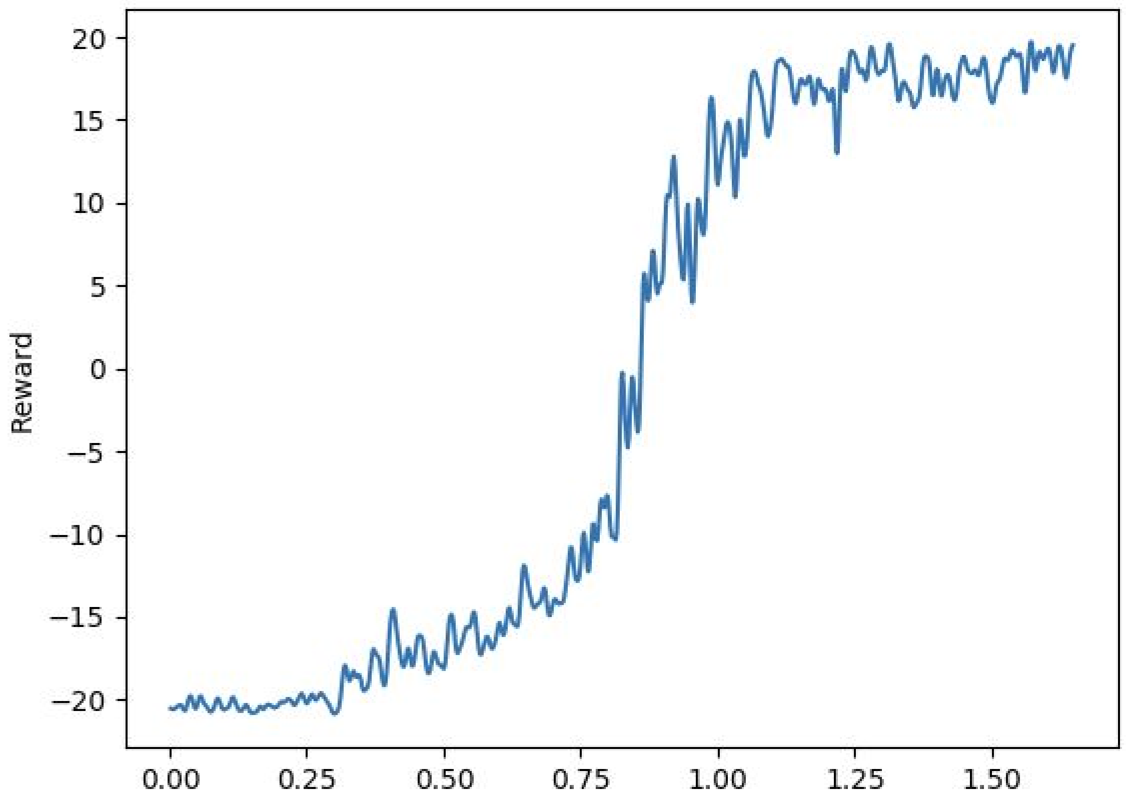
The model performed very well. Having a steep increase in rewards around 800K iterations. The model saturates just above 1 Million iterations and fluctuates around the 20s until the end. The maximum reward was 21 for some cases and the latest average reward was 20.

MODEL 2:

Looking at the trends of Model 1, I decided to reduce the number of iterations to 1.5 Million where the last model saturated. I also reduced the length of the replay memory to see if it had any effect on the training. I decided to keep Gamma at 0.99.

Gamma : 0.99
Number of frames: 1.5 Million
Length of Replay Memory: 150,000

The trend of the Rewards while training is as follows:



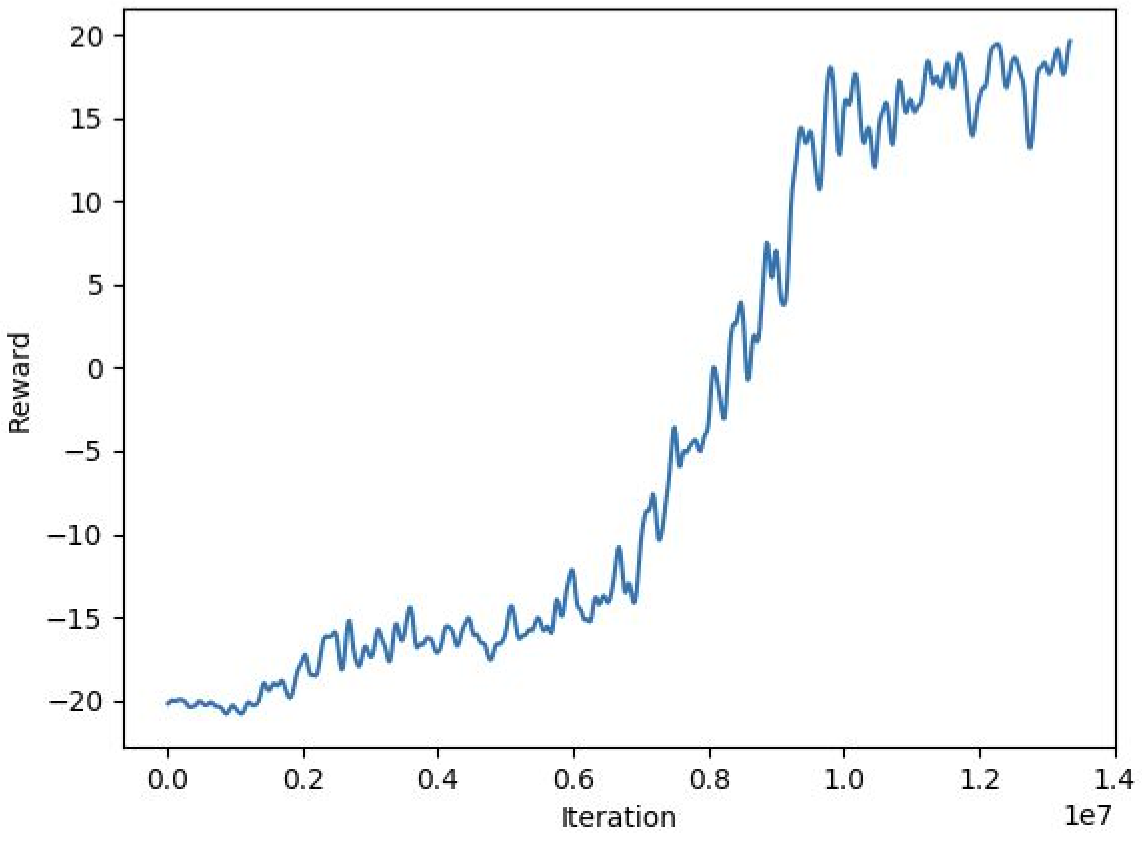
Maximum reward: 21
Average last reward: 19.7

The trend of the rewards for this model was similar to the first model. The main similarities are the time at which there is a steep jump in the average reward. However, the second model seemed to reach saturation faster than the previous model. This model did not fluctuate that often as compared to the previous one.

MODEL 3:

Gamma : 0.98
Number of frames: 1.5 Million
Length of Replay Memory: 200,000

The trend of the Rewards while training is as follows:



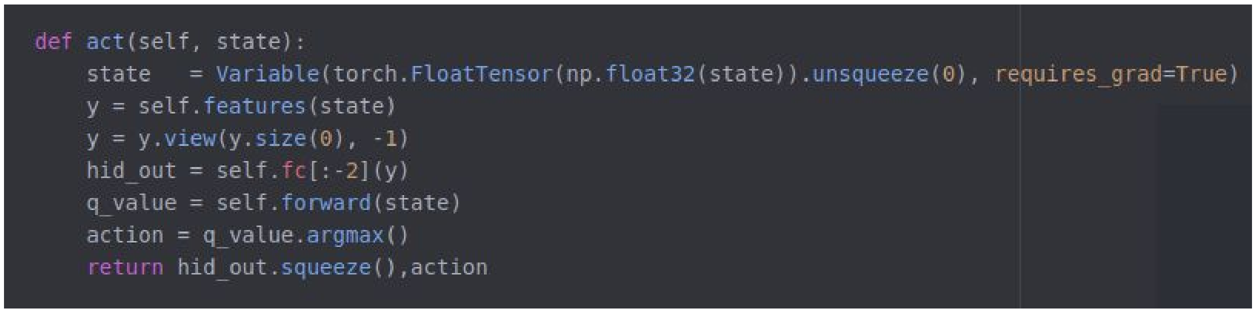
For the 3rd and final Model. I finally decided to tweak the Gamma to 0.98. I kept the number of iterations as 1.5M similar to Model 2 and increased the length of Replay Memory to 200K.

One noticeable difference between this model and the previous models is that, there is no steep increase in reward like the previous two. The increases is rather gradual, but accelerated around the same time that the steep increase was noticed in the former ones. This can be explained with the reduction of gamma, which encourages the model to explore more. Another noticeable difference is the variance of the rewards over the whole training process. As we can clearly see, the fluctuations of the reward throughout training is the most in this model compared to the previous 2.

Maximum reward: 21
Average last reward: 20.5

PART -2:

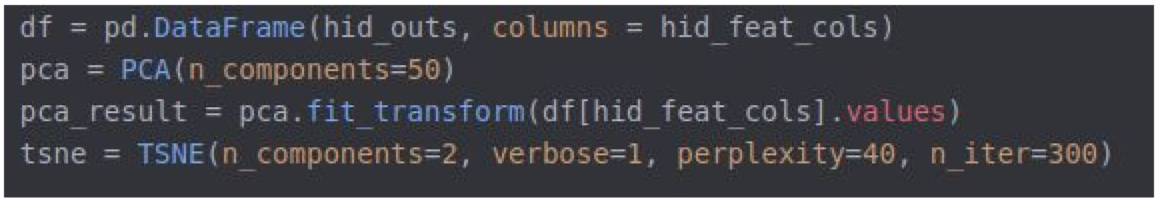
Qa: Evaluate your trained DQN in Part 1 on (e.g randomly picked 1000) frames and collect the 1000 × 512 features as well as various side information.



```
def act(self, state):
    state   = Variable(torch.FloatTensor(np.float32(state)).unsqueeze(0), requires_grad=True)
    y = self.features(state)
    y = y.view(y.size(0), -1)
    hid_out = self.fc[:-2](y)
    q_value = self.forward(state)
    action = q_value.argmax()
    return hid_out.squeeze(),action
```

Qb: Perform manifold-based dimensionality reduction on the features:

I chose to perform t-SNE (Stochastic Neighbor Embedding) to embed the 1000 X 512 features of the hidden layer to a 2-D space. Before this, I used PCA (Principal Component Analysis) to extract the first 50 principal components of the features. The first 50 principal components consisted of 87% of the variance of the features.

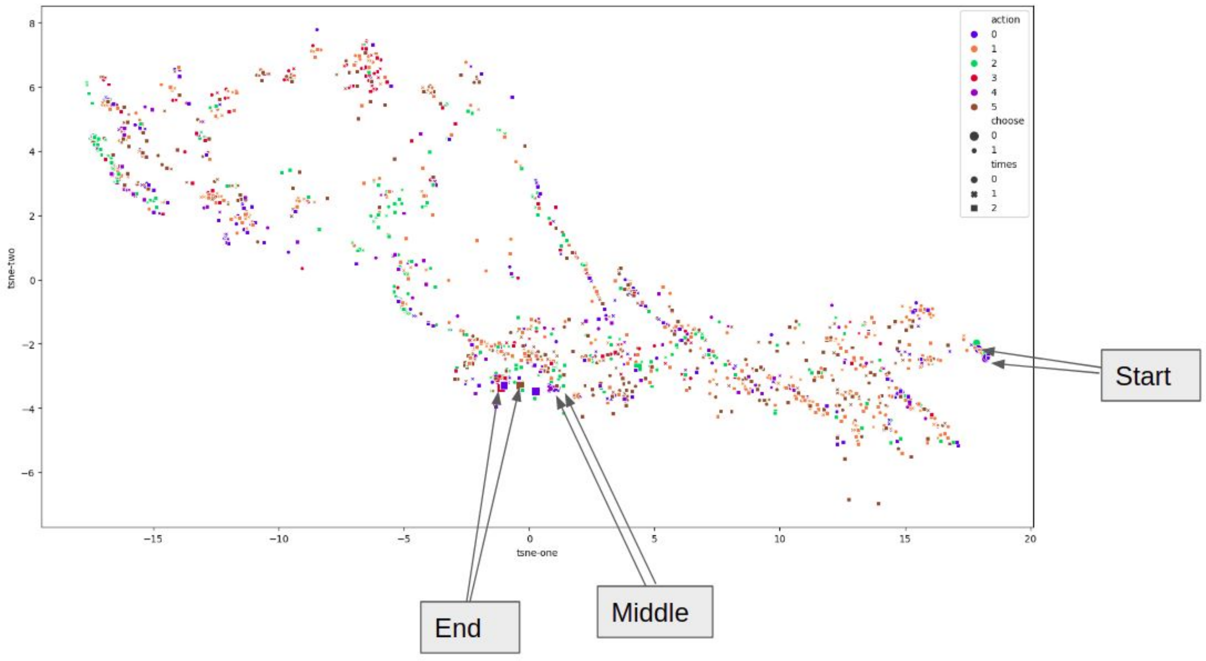


```
df = pd.DataFrame(hid_outs, columns = hid_feat_cols)
pca = PCA(n_components=50)
pca_result = pca.fit_transform(df[hid_feat_cols].values)
tsne = TSNE(n_components=2, verbose=1, perplexity=40, n_iter=300)
```

Qc: Analyze the embedding based on two kinds of your collected side information:

I chose to analyze the embedding based on **Time of Action** and **Action** given a particular state. The circles represent the Actions that were done at the beginning of the episode. The Crosses depict the actions done at the middle of the episode. Finally, the Boxes represent the actions done at the end of the episode.

The Hue of the points were based on the 6 possible Actions that can be done. First, when I just plotted the actions without associating it with the Time of Action, I noticed that the actions do not form clusters. This could be explained by the fact that, Pong is a two player game. The action of the agent depends upon the position of the opponent. Also, as there are only limited actions, they cannot be generalized for certain situations.



Therefore, I decided to include the Time of Action as well. Now there is a fairly noticeable pattern in the embedding. Some of the point at the Beginning, Middle, and End of an episode are chosen and enlarged in the picture above. As it is clearly seen, these actions are very close to each other. This confirms our hypothesis. In the particular stages of the episode, the position of the opponent does not vary much. Hence, the action chose also does not vary by much given a particular state.